UNIVERSITY OF WATERLOO COMPUTER SCIENCE DEPARTMENT

Direct Dynamic Structures for Some Line Segment Problems

Gaston H. Gonnet
J. Ian Munro
Derick Wood

CS-82-27

August, 1982

DIRECT DYNAMIC STRUCTURES FOR SOME LINE SEGMENT PROBLEMS[1)]

by

Gaston H. Gonnet, J. Ian Munro, and Derick Wood[2)]

(1) Work carried out under Natural Sciences and Engineering Research Council of Canada Grants, Nos. A-3353, A-7700 and A-8237.

(2) Department of Computer Science
University of Waterloo,
Waterloo, Ontario N2L 3G1
Canada

Abstract

We introduce simple direct dynamic binary tree structures for two line segment problems. Both structures provide for $O(\log_2 n)$ updating and querying and can be based on any balanced class of binary search trees, for example AVL-trees. The measure tree solves a non-decomposable searching problem, while the stabbing tree improves the known updating and querying time for its corresponding application.

1. Introduction

Our general interest in this paper is that of dynamic problems in computational geometry. Such issues arise, for example in the design of VLSI circuitry where problems of layout deal with the manipulation of rectangles whose sides are parallel to the coordinate axes in 2-space. The end product of such manipulations, a mask for a layer of a chip, is the specification of the region of the plane covered by at least one of the rectangles. In such applications, however, it is important to realize that not only must computation of a layout be performed efficiently, but that changes to the circuit are bound to be made during the design process. Hence modifications to the structure must be performed at modest cost.

The specific problems solved in this paper deal with keeping track of overlapping lines in a dynamic environment. The ultimate goal is the application of methods of the type developed here to problems of the type suggested above. Indeed the technical origin of our problem arises from such a problem, partially solved by Bentley (1977), and van Leewen and Wood (1981), namely:

Given n rectilinearly-oriented rectangles in 2-space compute the area of the plane that they cover, that is their measure.

Their algorithm uses the sweeping line paradigm to reduce the problem to a dynamic one-dimensional measure subproblem, namely:

Design a data structure in which line segments can be inserted and deleted in $O(\log_2 n)$ time and which gives the measure of the current line segments, that is the total length of the portions of the line which are covered, in $O(\log_2 n)$ time.

In both cases the same tree structure, the segment tree introduced by Bentley (1977), is used. However this data structure is only semi-dynamic, in that its structure does not change during updates. This is because the line segments to be inserted and deleted are known in advance and because the segment tree cannot easily support arbitrary updates.

This work raises the natural question:

Does there exist a structure which supports arbitrary updating of line segments and querying of the current measure in $O(\log_2 n)$ time.

In Section 2 we introduce the measure tree, to solve this problem which only requires $O(n)$ space and constant time for the current measure query, as well as $O(\log_2 n)$ for updates. Surprisingly the underlying structure can be any class of balanced binary search trees. For convenience we consider AVL-trees. It should be noted that the measure problem is not decomposable, cf. Bentley (1979).

In Edelsbrunner, van Leeuwen, Ottmann and Wood (1981) an algorithm to find the connected components of a set of rectilinearly-oriented rectangles, requires the following stabbing problem to be solved, namely:

Design a data structure in which line segments can be inserted and deleted in $O(\log_2 n)$ time and which gives the stabbing number of a query point x with respect to the current line segments in $O(\log_2 n)$ time.

The stabbing number of a point x with respect to n line segments is the number of line segments containing, or stabbed by, x.

In Section 3 we introduce the stabbing tree to solve this problem in the specified time bounds using $O(n)$ space. The stabbing problem unlike the measure problem is decomposable, see Bentley (1979), but general dynamization techniques, see Bentley and Saxe (1980) and van Leeuwen and Overmars (1981), do not yield such tight bounds.

Although these direct dynamic structures are of interest in their own right, the proofs that the structures can support rebalancing have wider application and interest. The basic tool for proving that rebalancing can be carried out efficiently in the measure and stabbing tree is a Reconstruction Lemma, which states that the additional information at a node in such a tree can be recomputed from that of its sons.

Before introducing these two structures we recall some basic notation.

Let T be a binary tree and u a node in T, then by T(u) we denote the subtree rooted at u, by λu the left son of u and by ρu the right son of u.

In the following sections a line segment L is specified by its two endpoints, $L = [x_1, x_2]$, where $x_1 < x_2$, denoting a closed interval of the line. For convenience we will assume the endpoints of all line segments to be unique throughout the paper. This assumption does not affect the results, it makes their presentation simpler.

2. Computing the Measure

Given n regions R_i, of d-space, then the measure of $R_1 \cup R_2 \cup \ldots \cup R_n$ is the fair d-volume, where the regions are interpreted as point sets. By fair d-volume we mean that for overlapping regions their region of overlap is only counted once. For 2-space we have the area covered by the regions. When each of the R_i is a one-dimensional line segment, then their measure is the total length of the portions of the line which are covered by at least one of the R_i. The problem of computing the measure in this case was posed and solved by Klee (1977), while Fredman and Weide (1978) proved that Klee's algorithm was optimal. When the R_i are rectilinearly-oriented rectangles in 2-space, an optimal algorithm to compute their measure is provided by Bentley (1977), see also van Leeuwen and Wood (1981), who also provide efficient algorithms for computing the measure of d-ranges in d-space.

However in each of these investigations only the off-line or static problem, posed above, is solved. The on-line or dynamic computation of the measure has not been tackled. In this section we consider the dynamic measure problem for line segments in 1-space, that is arbitrary sequences of: insert a line segment; delete a line segment; and query the current measure are allowed. We are able to show that each of the operations can be supported in $O(\log_2 n)$ time in the worst case in a data structure requiring $O(n)$ space, where the data structure currently contains n line segments. In fact querying the measure requires only constant time, as we shall see. It is important to realize that the measure problem is not decomposable, see Bentley (1979), Bentley and Saxe (1980) and van Leeuwen and Overmars (1981), and hence the dynamization paradigm is inapplicable.

We will construct such a data structure, <u>the dynamic measure tree</u>, in three stages. First we describe the <u>static measure tree</u>, second we describe how an insertion and deletion can be carried out in such a tree T, in O(height(T)) time, and third we describe how a single rotation of a node in T can be carried out in constant time.

If the measure tree is also an AVL-tree, Adelson-Velskii and Landis (1962), then the three stages outlined above ensure that the AVL-insertion and AVL-deletion algorithms can be implemented without any deleterious effects. Moreover as the root of the measure tree contains the current measure we will have proved the following required theorem.

<u>Theorem 2.1.</u>

The dynamic one-dimensional measure problem can be solved using $O(n)$ space for n currently active line segments, $O(\log_2 n)$ time for an insertion or deletion of a line segment and $O(1)$ time for the current measure query.

2A. <u>The Static Measure Tree</u>

The measure tree we will describe is a binary search tree for the endpoint values of the line segments, together with eight further fields. Initially we assume that all endpoint values are distinct. More precisely, the fields for each node u in a measure tree are:

(i) value(u) - either a left or right endpoint,

(ii) which(u) - whether the endpoint is a left or right endpoint,

(iii) other(u) - the value of the partner (endpoint),

(iv) min(u) - the minimum value in T(u),

(v) max(u) - the maximum value in T(u),

(vi) leftmin(u) - the minimum left endpoint value with respect to the line segments represented in T(u). Note that leftmin(u) can be outside T(u).

(vii) rightmax(u) - the maximum right endpoint value with respect to the line segments represented in T(u). Note that rightmax(u) can be outside T(u).

(viii) submeasure(u) - the measure of the line segments represented in T(u) by at least one endpoint, but only with respect to the interval [min(u), max(u)].

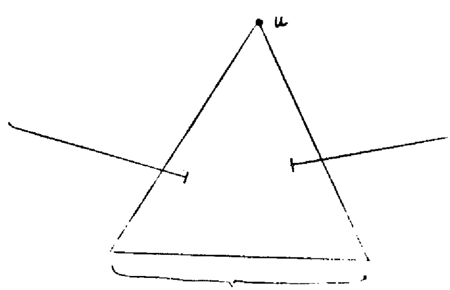


Figure 2.1

Of these fields it is only the last three that possibly require further clarification.

Submeasure(u), see Figure 2.1, can be viewed as the "blinkered" measure determined by T(u). In other words only the portion of the interval [min(u), max(u)], which is covered by the line segments in T(u) is considered.

The other two fields leftmin and rightmax, are necessary to Stage 3. Note that when all left endpoints in T(u) have partners in T(u), then leftmin(u) reverts to min(u), since this then represents the minimum left endpoint value represented in T(u). A similar remark holds for max(u) and rightmax(u).

Observe that n line segments require 2n nodes and each node requires constant space, hence O(n) space is required in total. Furthermore if u is the root of such a tree T, then submeasure(u) is indeed the measure of the line segments in T. Therefore only O(1) time is required to answer a measure query.

2B. Insertion and Deletion

In order to demonstrate how insertion is carried out we need the following lemma.

Lemma 2.1 The Reconstruction Lemma

Let T be a measure tree and u be any node in T. Then the fields (iv) -

(viii) of u can be reconstructed from the first two fields of u together with the fields of u's left and right sons.

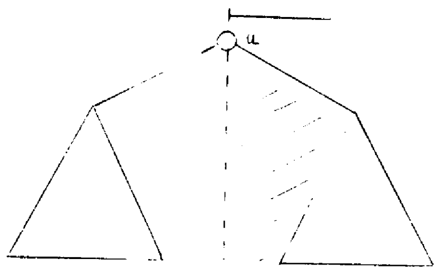


Figure 2.2

<u>Proof</u>: (iv) Clearly

$$\min(u) = \begin{cases} \text{value}(u), & \text{if } \lambda u \text{ is empty,} \\ \min(\lambda u), & \text{otherwise} \end{cases}$$

(v) Is similar to (iii).

(vi) Clearly

$$\text{leftmin}(u) = \begin{cases} \min\{\text{value}(u), \text{leftmin}(\lambda u), \text{leftmin}(\rho u)\} & \text{if which}(u) = \text{left,} \\ \min\{\text{other}(u), \text{leftmin}(\lambda u), \text{leftmin}(\rho u)\} & \text{otherwise,} \end{cases}$$

If λu or ρu is empty the corresponding term is ommitted.

(vii) Is similar to (vi).

(viii) Assume which(u) = left; the case which(u) = right is symmetric. There are three cases to consider:

1. (leftmin(ρu) = min(ρu) or leftmin(ρu) = value(u)) and rightmax(λu) = max(λu). See Figure 2.2. In this case the only contribution to the gap between max(λu) and min(ρu) is from value(u) itself.

$$\text{submeasure}(u) = \begin{cases} \text{submeasure}(\lambda u) + \text{submeasure}(\rho u) + \min(\rho u) - \text{value}(u) \\ \qquad \text{if other}(u) \text{ is in } T(\rho u), \\ \text{submeasure}(\lambda u) + \max(\rho u) - \text{value}(u), \text{ otherwise} \end{cases}$$

2. rightmax(λu) $\neq$ max(λu).

   The gap is completely covered, hence

$$\text{submeasure}(u) = \begin{cases} \text{submeasure}(\lambda u) + \text{submeasure}(\rho u) + \min(\rho u) - \max(\lambda u) \\ \qquad \text{if other}(u) \text{ is in } T(\rho u), \\ \text{submeasure}(\lambda u) + \max(\rho u) - \max(\lambda u), \text{ otherwise} \end{cases}$$

3. leftmin(ρu) $\neq$ min(ρu) and leftmin(ρu) $\neq$ value(u) and not case 2.

   The gap is completely covered.

$$\text{submeasure}(u) = \begin{cases} \text{submeasure}(\lambda u) + \text{submeasure}(\rho u) + \min(\rho u) - \text{value}(u) \\ \qquad \text{if other}(u) \text{ is in } T(\rho u), \\ \text{submeasure}(\rho u) + \min(\rho u) - \min(\lambda u), \text{ otherwise} \end{cases}$$

Thus in all five cases the reconstruction can be carried out ∎

We utilize Lemma 2.1 during both insertion and deletion. First consider insertion.

Insertion

Given a measure tree T with the information described above and a line segment $L = [x_1, x_2]$ we search T with both x_1 and x_2 simultaneously. This search describes a forked path in T, see Figure 2.3(a), which may be degenerate, see Figure 2.3(b).

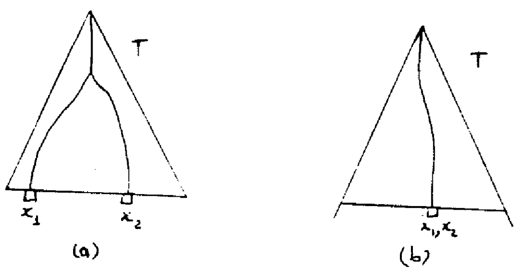
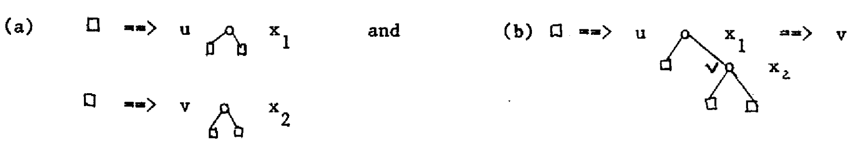


Figure 2.3

In both cases we add x_1 and x_2 to T, viz.



and in both cases we initialize the fields of u and v to their appropriate values.

(a.i) which(u) := left; other(u) := x_2; min(u) := max(u) := x_1;
leftmin(u) := x_1; rightmax(u) := x_2; submeasure(u) := 0;

(a.ii) which(v) := right, other(v) := x_1; min(v) := max(v) := x_2;
leftmin(v) := x_1; rightmax(v) := x_2; submeasure(v) := 0;

(b) as for (a) except that:

max(u) := x_2; submeasure(u) := $x_2 - x$

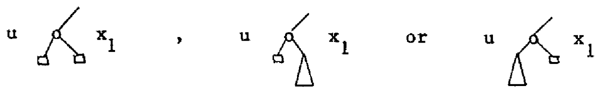
Now the search path(s) in T are retraced and at each revisited node Lemma 2.1 is invoked to re-calculate the 6 fields. In case (a) the recalculation at the "fork" node is only carried out after both its sons have been revisited. Thus the time taken for the insertion is proportional to the length of the search path(s) and hence is O(height(T)) in the worst case.

Deletion

Given a measure tree T and a line segment L = $[x_1, x_2]$ we search T with both x_1 and x_2 simultaneously. Again we obtain a forked path similar to that of Figure 2.3(a), except that the termination nodes contain x_1 and x_2.

As is usual in deletion we distinguish between x_1 and x_2 appearing in frontier nodes and internal nodes. Consider x_1 only in the following:

1. x_1 is a frontier node u, that is u is one of:



In each case u can either be replaced by the empty subtree or by its non-empty subtree. The recomputation of each field of every node on the search path is now carried out based on Lemma 2.1.

(2) x_1 is in an internal node u, that is u is:

In this case, we carry out the standard technique, namely we delete min(ρu) from T(ρu) and replace x_1 by min(ρu). The first stage is a type(1) deletion, and therefore the removal of min(ρu) is straightforward and also the fields on the leftmost path in T(ρu) can be recomputed. Second on replacing x_1 by min(ρu), the recomputation of the fields in u and its predecessors can also be carried out via Lemma 2.1.

Hence both types of deletion can be carried out in time proportional to the height of T.

2C. Rotations

Most balanced binary tree schemes make use of single and double rotations to rebalance a tree after an insertion or deletion. Now double rotations consist of two successive single rotations, hence it is only necessary to consider single rotations.

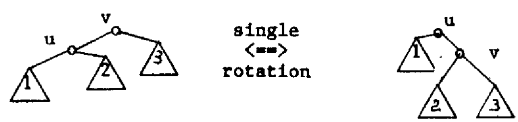


Figure 2.4

Reading Figure 2.4 from left to right illustrates a single rotation of v, while from right to left it illustrates a single rotation at u. We only consider the former since the latter follows symmetrically. We will subscript the u and v with a(fter) and b(efore). Observe immediately that

$$\min(u_a),\ \max(u_a), \ldots,\ \text{submeasure}(u_a) =$$

$$\min(v_b), \ldots,\ \text{submeasure}(v_b), \text{ respectively}$$

since the values in $T(v_b)$ are the same as those in $T(u_a)$.

However $\min(v_a), \ldots,$ submeasure(v_a) are not the same as $\min(u_b), \ldots,$ $\ldots,$submeasure(u_b), but in this case we can compute them by invoking Lemma 2.1. Hence we have demonstrated that a rotation in a measure tree will give a new measure tree and furthermore this transformation can be effected in O(1) time. This completes the proof of Theorem 2.1.

3. Computing the Stabbing Number

We say that a point x, (in 1-space) stabs a line segment $L = [x_1, x_2]$ in 1-space, if $x_1 \leq x \leq x_2$. Given n line segments in 1-space, and a point x in 1-space, then the stabbing number of x with respect to the n line segments is the number of line segments stabbed by x. Clearly these concepts can be generalized to arbitrarily dimensioned spaces, but this is not our concern here. As in Section 2 we are concerned with the on-line or dynamic stabbing number problem, that is:

Construct a data structure to maintain line segments, which allows insertion and deletion of line segments and for an arbitrary point x determines its stabbing number, all in $O(\log_2 n)$ time and $O(n)$space, when the data structure currently holds n line segments.

We parallel the approach taken in Section 2 by first presenting a static structure, the (static) stabbing tree, second showing how insertion and deletion can be carried out and third, how rotations can be achieved.

In the present case we are dealing with a decomposable problem a la Bentley (1979), however our direct approach is of interest not only in its own right, but also because it provides the best known bounds to date. We will prove the following:

Theorem 3.1

The dynamic one-dimensional stabbing problem can be solved using O(n) space for n currently active line segments and $O(\log_2 n)$ time for updating with a line segment and querying with a point for its stabbing number.

3A. The Static Stabbing Tree

The static stabbing tree is basically a binary search tree for the 2n endpoint values of the given line segments ($n \geq 1$). Again for simplicity we assume these values are distinct. However the nodes have three additional fields, all of which we now specify: Let u be a node in such a tree T, then:

(i) value(u) - the endpoint value,

(ii) which(u) - whether value(u) is a left or right endpoint value,

(iii) other(u) - the position of the partnering endpoint in T.

(iv) balance(u) - the difference between the number of left endpoint values and the number of right endpoint values in T(u).

To answer a stabbing query we have:

The Stabbing Query Algorithm

Given: A stabbing tree T with root u, a point query x, and two variables leftstab and rightstab, both initially zero.

1. u is a leaf: The stabbing number is leftstab if $x < \text{value}(u)$, rightstab if $x > \text{value}(u)$ and max(leftstab, rightstab) if $x = \text{value}(u)$.

2. u is not a leaf:

   2.1 x < value(u): Let rightstab be leftstab + balance(u) and recursively call the algorithm with u equal to λu.

   2.2 x > value(u): Let leftstab be rightstab - balance(u) and recursively call the algorithm with u equal to ρu.

   2.3 x = value(u): The stabbing number is max(leftstab + balance(λu), rightstab + balance(ρu)).

End of stabbing query algorithm.

This gives rise to immediately to:

Lemma 3.1

Given a stabbing tree T and a point query x, the stabbing query algorithm returns the stabbing number of x with respect to the line segments in T in O(height(T)) time.

Proof: Clearly the recursive stabbing algorithm takes O(height(T)) time, hence it only remains to demonstrate its correctness. However this follows immediately from the recursive invariant:

> On entry to the stabbing query algorithm at node u in the stabbing tree T, leftstab and rightstab are the stabbing depths immediately to the left and right, respectively, of T(u) with respect to T.

See Figure 3.1. ∎

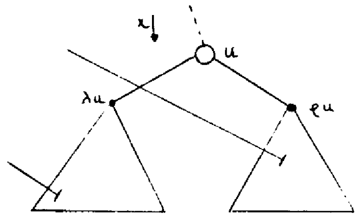


Figure 3.1

3B. Insertion and Deletion

Examination of the four fields in each node of a stabbing tree T shows that only the field balance(u) needs to be updated along the search path, when inserting or deleting a line segment $L = [x_1, x_2]$. A search for both x_1 and x_2 is carried out simultaneously yielding a forked path in T, which may degenerate, c.f. Section 2B and Figure 2.3. Consider the left fork of such a forked path. The subtree T(u) of every node u on this fork contains only the left endpoint of L. Similarly subtrees rooted on the right fork contain only the right endpoint of L. On the left fork balance(u) is incremented or decremented by one for an insertion or deletion, respectively. Whereas on the right fork balance(u) is decremented or incremented by one for an insertion or deletion, respectively. The balance at each of the nodes on the initial (unforked) portion of the search path is, of course unchanged.

3C. Rotations

The balance of a node is easily reconstructable since balance(u) = balance(left(u)) + balance(right(u)) + δ, where $\delta = 1$ if which(u) = left and

-1 otherwise. Hence we have:

Lemma 3.2 The Reconstruction Lemma

Let T be a stabbing tree, u be any nonleaf node in T, then the balance at u can be recomputed from the balance of its sons.

And this completes the Theorem.

References

Bentley, J.L. Solutions to Klee's rectangle problems. Unpublished manuscript, Carnegie-Mellon University (1977).

Bentley, J.L. Decomposable searching problems. Information Processing Letters 8 (1979), 244-251.

Bentley, J.L., and Saxe, J.B. Decomposable searching problems I: static-to-dynamic transformation. Journal of Algorithms 1 (1980), 301-358.

Edelsbrunner, H., van Leeuwen, J., Ottmann, Th., and Wood, D. Connected components of orthogonal geometric objects. McMaster University Technical Report 81-CS-04, 1981.

Fredman, M., and Weide, B. On the complexity of computing the measure of $U[a_i, b_i]$. Communications of the ACM 21 (1978), 540-544.

Klee, V. Can the measure of $U[a_i, b_i]$ be computed in less than O(nlogn) steps? American Mathematical Monthly 84 (1977), 284-285.

van Leeuwen, J.L., and Overmars, M.H. The art of dynamizing. Proceedings of Mathematical Foundations of Computer Science 1981 (ed., J. Gruska and M. Chytil), Springer-Verlag Lecture Notes in Computer Science 118 (1981), 121-131.

van Leeuwen, J.L., and Wood, D. The measure problem for rectangular ranges in d-space. Journal of Algorithms 2 (1981), 282-300.

August 11, 1982
UCS4/DW.1/A/DW3.1/dm