A Space-Efficient Algorithm for Segment Intersection

Eric Y. Chen* Timothy M. Chan*†

Abstract

We examine the space requirement for the classic line-segment intersection problem. Using so-called implicit data structures, we show how to make the standard sweep-line algorithm run in $O((n+k)\log^2 n)$ time with only $O(\log^2 n)$ extra space, where n is the number of line segments and k is the number of intersections. If division is allowed and input can be destroyed, the algorithm can run in $O((n+k)\log n)$ time with $O(1)$ extra space.

1 Introduction

Recently, Brönnimann et al. [4] revisited a basic geometric problem (namely, *planar convex hulls*) from the point of view of minimizing space: the input is initially given in an array; algorithms can modify the array but are allowed only a small amount of extra memory (usually $O(1)$ or $O(\text{polylog}\, n)$). In the "non-destructive" model, we insist further that the final array holds a permutation of the original input elements. Such *space-efficient* algorithms are analogous to "in-place" sorting algorithms (e.g., heapsort). They are important in practice, because they enable larger problem instances to be solved in main memory.

In this paper, we reconsider another basic geometric problem, *segment intersection*, from the same point of view. Given an array of n line segments in the plane (each stored as a record of four coordinates), we would like to print all k pairs of intersections; the output need not be stored in memory. The brute-force method requires only $O(1)$ extra space but runs in $O(n^2)$ time. A practical output-sensitive algorithm running in $O((n+k)\log n)$ time was first obtained by Bentley and Ottmann [3] using the sweep-line technique. Time-optimal algorithms, with $O(n\log n + k)$ complexity, were later given by Chazelle and Edelsbrunner [6] and Balaban [2]. Chazelle and Edelsbrunner's algorithm required $O(n+k)$ extra space, whereas Bentley and Ottmann's and Balaban's required $O(n)$ extra space.

In contrast to Brönnimann et al. [4], we face greater challenges in making these segment intersection algorithms space-efficient. Bentley and Ottmann's sweep-line algorithm [3], for example, requires the maintenance of more than one tree structures: a balanced search tree and a priority queue. The search tree alone poses problems, as storing even a single pointer per segment would require $\Omega(n)$ extra space!

We observe that techniques from the area of *implicit data structures* can help in meeting these challenges. For example, Munro [11] had given search trees that used only $O(\log^2 n)$ extra space and supported query and update operations in $O(\log^2 n)$ time. The main idea is to encode pointers implicitly by permuting elements within blocks of logarithmic size.

In our application, we need to further "overlay" a priority queue with the implicit tree in a way that does not introduce extra space. We show that such a combination is possible with Munro's structure. The result is a segment intersection algorithm with $O((n+k)\log^2 n)$ time and $O(\log^2 n)$ space under the non-destructive model.

We also notice a space-saving trick for storing line segments under the destructive model if division is allowed (or alternatively if segments are given in slope-intercept form). With this trick, our algorithm can be simplified and made to run in $O((n+k)\log n)$ time and $O(1)$ space. We have implemented this version of the algorithm and report on the test results. Because of the use of division, the implementation is not guaranteed to be robust, though.

2 The Sweep-Line Algorithm

Our algorithm follows Bentley and Ottmann's approach [3, 7] and keeps all segments intersecting a vertical sweep line ordered from top to bottom in a search tree T. As the sweep line moves from left to right, the next event (change to T) occurs when the sweep line encounters a left endpoint, a right endpoint, or the intersection of an adjacent pair along the sweep line. This next event can be determined using a priority queue Q; we alter the usual description of Q here, with space considerations in mind:

First, by pre-sorting the given array according to the x-coordinates of the left endpoints (e.g., by heapsort), we can easily identify the next left-endpoint event. So, we only need Q to store right-endpoint and intersection events.

For each segment ℓ intersecting the sweep line, define *ℓ's event* to be the intersection of ℓ with its successor

*School of Computer Science, University of Waterloo, Waterloo, ON N2L 3G1, Canada, {y28chen,tmchan}@uwaterloo.ca

†Supported by an NSERC Research Grant and the Premier's Research Excellence Award of Ontario.

in T, if the intersection exists and is to the right of the sweep line; otherwise, define ℓ's event to be the right endpoint of ℓ. Clearly, the leftmost of these segments' events gives us the next right-endpoint/intersection event. We keep all segments' events in Q, with x-coordinates as priorities.

Let ℓ^+ and ℓ^- denote the successor and predecessor of ℓ in T. The algorithm can be expressed as follows:

Initialize T and Q
Repeat:
  Let e be the leftmost among the events in Q and the next left endpoint
  Move the sweep line to e
  Case 1: e is the left endpoint of a segment ℓ
    Add ℓ to T
    Add ℓ's event to Q
    Update ℓ^-'s event in Q
  Case 2: e is the right endpoint of a segment ℓ
    Remove ℓ from T
    Remove e from Q
    Update ℓ^-'s event in Q
  Case 3: e is the intersection of segments ℓ and ℓ^+
    Print e
    Switch the order of ℓ, ℓ^+ in T
    Update ℓ^-'s, ℓ's, and ℓ^+'s events in Q

Naively, T and Q take up $O(n)$ extra space. We describe a more space-efficient way to handle these data structures, following the rough organization of the array as depicted in Figure 1.

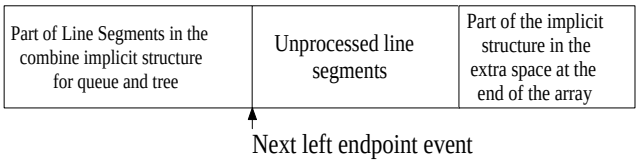

Figure 1: The memory layout of our algorithm.

3 The Implicit Search Tree

We use Munro's implicit tree structure [11] to store T. His structure is similar to standard balanced search trees but avoids extra space for pointers. It has two parts: a balanced binary tree and an auxiliary set of $s-1$ lists. Each tree node, called a *tree block*, holds exactly s elements rather than one. Elements within each block are kept basically sorted. All elements in one tree block must be greater than all elements another block or vice versa. To support insertion/deletion of a single element, we allow gaps between successive tree blocks. Gaps may have size ranging from 1 to $s-1$. Gaps of the same size are stored in a common list. Each list is divided in *list blocks* of size exactly s and resides in the given array, with the exception of the list's head block (which is not necessarily full).

In addition to the usual left/right/successor/predecessor pointers, each tree block has a pointer to a list block that stores the gap between the tree block and its successor. In an insertion/deletion, a gap size is incremented/decremented, and consequently a gap has to be removed from one list and moved to another; this can be accomplished by manipulating $O(s)$ elements and $O(1)$ pointers. When a gap reaches size s, a list block is deleted and a new tree block is formed. Further details of the query and update algorithm can be found in Munro's paper [11].

We can avoid storing the necessary pointers (array indices) for each block by encoding them in a permutation of the block's elements. Specifically, we can represent a 0 or 1 by placing a pair of adjacent elements in order or not in order. We can thus represent $s/2$ bits with a block of s elements, as suggested in Figure 2. Setting $s = c \log n$ for a sufficiently large constant c is good enough for the implicit structure. Only $O(s^2)$ extra space is needed for the head blocks of the $s-1$ lists. Each query/update requires $O(\log \frac{n}{s})$ pointer operations and takes $O(s \log \frac{n}{s})$ time.

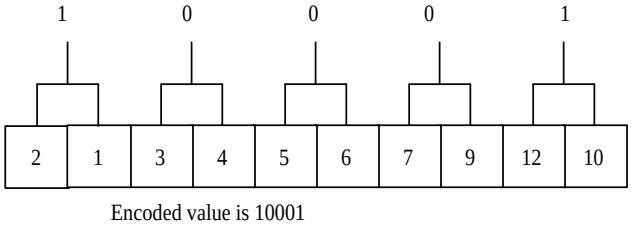

Figure 2: The bit-encoding strategy.

4 The Combined Priority Queue and Search Tree

We now show that the priority queue Q can also be embedded within the implicit tree structure T without using additional space. The key idea is to keep track of a priority value per block rather than per segment. Define a *tree block's event* to be the leftmost among the events of the segments in the block and the gap between the block and its successor. Clearly, the leftmost of the tree block's events gives us the leftmost of all segments' events. We keep a priority queue of the tree blocks' events, by encoding two more pointers per block (with a larger constant c). Within each tree block, we also keep pointers to the segment and successor defining the block's event.

Each update to a segment's event only affects a single block's event, which can be recalculated in $O(s)$ time (as the successors of the segments involved can all be identified in $O(s)$ time); the leftmost event in the

priority queue can be maintained by $O(\log \frac{n}{s})$ pointer operations and thus takes $O(s \log \frac{n}{s})$ time. Each insertion/deletion only changes a constant number of blocks/gaps in the implicit search tree; the leftmost event can again be maintained in $O(s \log \frac{n}{s})$ time. The cost per update/query thus remains $O(s \log \frac{n}{s})$.

5 Analysis

Our segment intersection algorithm uses only $O(s^2)$ extra space. Since there are $O(n+k)$ events, the total number of updates and queries performed by the algorithm is $O(n+k)$ and the overall running time is $O(s(n+k)\log \frac{n}{s})$. With $s = \Theta(\log n)$, the time bound is $O((n+k)\log^2 n)$ and the extra space used is $O(\log^2 n)$.

6 Implementation

In the destructive model, we observe a trick that eases implementation and improves the time bound at the same time. Convert each line segment to a 4-tuple consisting of the left/right x-coordinates, the slope, and the intercept. When the sweep line passes its left endpoint and the segment is being stored in T, its left x-coordinate is no longer needed. We can thus gain one free space per segment in the tree. With this trick, we can afford to use a sufficiently large constant as the block size and still have space to store the $O(1)$ pointers needed for each block. We can also bypass the bit-encoding/decoding strategy which makes programming difficult. With $s = O(1)$, the time bound is $O((n+k)\log n)$ and the extra space used is $O(1)$.

For the implementation, instead of using the more well-known AVL or red-black tree, we decide to adopt Andersson's balanced tree [1], which has simplicity as its main advantage (where the update procedures are expressed elegantly in terms of two operations called skew and split). We use doubles to store all values; we find that a block size of $s = 9$ is enough to keep the necessary pointers within the array.

Table 1 gives a comparison of the running time and space requirement of our version of the sweep-line algorithm with a version that uses regular data structures. The experiments were conducted on a PC with a Pentium 4 2.4 GHz processor, with randomly generated line segments. (More precisely, one of the endpoints is uniformly distributed inside the unit square, the direction is uniformly distributed as well, but the length is fixed at $1/\sqrt{n}$.)

The space-efficient version is slower, as can be expected, due to a larger constant factor in the big-O bound, but the implementation is still at a preliminary stage and we haven't explored ways to speed up the code yet.

n,k	Time (mSec)		Space (double)	
	Our Alg.	Old Alg.	Our Alg.	Old Alg.
1000,285	516	125	324	621272
10K,3K	3184	641	324	1374968
100K,32K	92375	7468	324	9225856
1M,319K	1197016	crash	324	n/a

Table 1: Experimental results.

7 Conclusions

We have shown how to obtain a space-efficient algorithm for the segment intersection problem by adopting implicit data structures and overlaying multiple ordered structures. This technique is quite general and might have applications to other problems in computational geometry, especially those solved by the sweep-line approach. One obvious candidate is *red/blue segment intersection* [5]. We are currently investigating space-efficient solutions to proximity problems such as planar Voronoi diagrams.

Faster implicit tree structures have recently been discovered by Franceschini et al. [8, 9, 10]. Another interesting question is whether it is possible to reduce our time and space bounds using these more complicated structures.

References

[1] Andersson, A. Balanced search trees made simple. In *Proc. 3rd Workshop on Algorithms and Data Structures* (1993), vol. 709 of *Lect. Notes in Comput. Sci.*, Springer, pp. 60–72.

[2] Balaban, I. J. An optimal algorithm for finding segments intersections. In *Proc. 11th Sympos. on Comput. Geom.* (1995), pp. 211–219.

[3] Bentley, J. L., and Ottmann, T. A. Algorithms for reporting and counting geometric intersections. *IEEE Trans. Comput. C-28* (1979), 643–647.

[4] Brönnimann, H., Iacono, J., Katajainen, J., Morin, P., Morrison, J., and Toussaint, G. Space-efficient planar convex hull algorithms. In *Proc. Latin American Theoretical Informatics* (2002), pp. 494–507. *Theoret. Comput. Sci.*, to appear.

[5] Chan, T. M. A simple trapezoid sweep algorithm for reporting red/blue segment intersections. In *Proc. 6th Canad. Conf. Comput. Geom.* (1994), pp. 263–268.

[6] Chazelle, B., and Edelsbrunner, H. An optimal algorithm for intersecting line segments in the plane. *J. ACM 39* (1992), 1–54.

[7] de Berg, M., van Kreveld, M., Overmars, M., and Schwarzkopf, O. *Computational Geometry: Algorithms and Applications*, 2nd ed. Springer-Verlag, 1998.

[8] Franceschini, G., and Grossi, R. Implicit dictionaries supporting searches and amortized updates in $O(\log n \log\log n)$ time. In *Proc. 14th ACM-SIAM Sympos. on Discrete Algorithms* (2003), pp. 670–678.

[9] Franceschini, G., and Grossi, R. Optimal worst-case operations for implicit cache-oblivious search trees. In *Proc. 8th Workshop on Algorithms and Data Structures* (2003). To appear.

[10] Franceschini, G., Grossi, R., Munro, J. I., and Pagli, L. Implicit B-trees: New results for the dictionary problem. In *Proc. 43rd IEEE Sympos. Found. of Comput. Sci.* (2002), pp. 145–154.

[11] Munro, J. I. An implicit data structure supporting insertion, deletion, and search in $O(\log^2 n)$ time. *J. Comput. Sys. Sci. 33* (1986), 66–74.