Applications of the Exponential Search Tree in Sweep Line Techniques

S. SIOUTAS, A. TSAKALIDIS, J. TSAKNAKIS. V. VASSILIADIS
Department of Computer Engineering and Informatics
University of Patras 2650 0Patras
GREECE

Abstract: - This paper refers to the «line segment intersection problem» using the sweep line technique. It describes how the Exponential search trees of Anderson can be used in this dynamic problem and how using this method is possible to achieve time O((n+s) √logn) maintaining the space cost linear. CSCC'99 Proc.pp.2261-2266

Key-words:- Computational Geometry, line segment intersection problem, sweep line technique, exponential search tree.

1. Introduction and Related Work

Consider the problem of reporting the intersections of n line segments in the plane. This problem is an excellent vehicle for introducing the powerful technique of plane sweep (fig. 1). The plane-sweep algorithm maintains an active list of segments that intersect the current sweepline, sorted from bottom to top by intersection point . If two line segments intersect, then at some point prior to this intersection they must be consecutive in the sweep list. Thus, we need only test consecutive pairs in this list for intersection, rather than testing all $O(n^2)$ pairs.

At each step the algorithm advances the sweep line to the next event: a line segment endpoint or an intersection point between two segments. Events are stored in a priority queue by their x-coordinates. After advancing the sweepline to the next event point, the algorithm updates the contents of the active list, tests new consecutive pairs for intersection, and inserts any newly-discovered events in the priority queue. For example, in fig.1 the locations of the sweepline are shown with dashed lines.

Bentley and Ottmann showed that by using plane sweep it is possible to report all s intersecting pairs of n line segments in O((n+s)logn) time [8]. If the number of intersections s is much less the $O(n^2)$ worst-case bound, then this is great savings over a brute-force test of all pairs. For many years the question of whether this could be improved to O(nlogn + s) was open , until Edelsbruner and Chazelle presented such an algorithm [9].

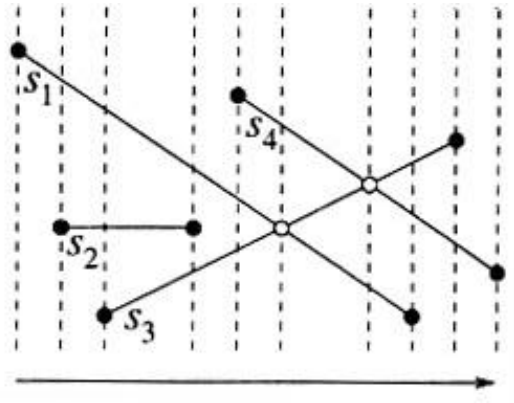

fig. 1

This algorithm is shown optimal with respect to running time because at least Ù(s) time is needed to report the result, and it can be shown that Ù(nlogn) time is needed to detect whether there is any intersection at all. However , their algorithm uses O(n + s) space. Clarkson and Shor presented a randomised algorithm with the same excepted running time but using only O(n) space [10]. The question if there is a deterministic O(nlogn + s) time and O(n) space algorithm for reporting line intersections is still an open problem. Based on [8] we present a very simple O((n+s) √logn) time and O(n) space algorithm using the Exponential Search tree. The Exponential Search trees [2] ar emultiway trees that answer efficiently queries in one-dimensional space. Their combination with a number of other techniques such as the Fusion technique [1], van Emde Boas trees [3] and perfect hashing [4], results in very fast searching structures.

2 Preliminary Data Structures

In this section we will briefly review the data structures which are used in our solution.

2.1 The Fusion tree

The Fusion tree is a static data structure which permits O(logN/loglogN) amortised time queries in linear space. This structure is used to implement a B-tree where only the upper levels in the tree contain B-tree nodes, all having the same degree (within a constant factor). At the lower levels, weight balanced trees are used. The amortised cost for searches and updates is O(logN/logd + logd) for any d = $O(w^{1/6})$. The first term corresponds to the number of B-tree levels and the second to the height of the weighted-balanced trees.

The Fusion tree has the following properties:

For any d, d = $O(w^{1/6})$, a static data structure containing d keys can be constructed in $O(d^4)$ time and space, such that it supports neighbour queries in O(1) worst-case time.

The main advantage of the fusion technique is that we can decide in time O(1) in which subtree to continue the searching by compressing the k-keys of every B-tree node using w - bit words.

It's necessary for the reader to be familiar with the details of the fusion algorithms of Willard concerning the costs of insertions and deletions in such a tree in full dynamic case. So , we refer two very important Lemmas [5].

LEMMA 1: Consider a B- tree whose internal nodes have parity between B/8 and B , whose root has parity between 2 and B , and whose leaves store the data and all have the same depth. Suppose that insertions and deletions in such a tree of height h will have an O(h) cost when no splits or merges occur, and the costs of splits and merges is bounded by O(Bh). Then regardless of the details of the structure of the node v (for example it could be a q-heap), it is possible to devise an insertion and deletion algorithm for this tree that runs in amortised time O(h).

Proof sketch: Without loss of generality , we may assume B>8. Consider the natural B-tree insertion / deletion algorithm that merges a non-root internal node v with its sibling w if v' s parity is less than B/8 , splits a node into two equal halves whenever its parity exceeds B (sometimes a merge will immediately trigger a split), and which makes the child of tree root into the new root if the preceding operations caused the old root to have only one child. It is easy to devise an accounting function that shows there will be only an amortized number of O(1/B) splits and merges in a tree of height h (essentially because nodes of height j will have an amortised frequency of $O[(8/B)^{j+1}]$ of splitting and merging). Hence the split and merge operations will have an O(h) amortised cost. This shows that the total cost of insertions and deletion is also O(h), since splitting and merging are the only potentially costly operations. (Lemma 1 also holds if splits and merges have a cost proportional to the number of leaf descendants (or indeed an O(Bh√Polynomial(j)) cost for a node of height j in a tree of height h), but the latter two are not relevant to our present discussion. Multi _branching B-trees with such properties were presented in [6].)

LEMMA 2: For sets of arbitrary cardinality M, pre-computed tables of size o(N) also make it possible to develop variants of q*-heaps that have a worst-case time O(1+logM/loglogN) for insertion, deletion and retrieval operations, i.e. the data structure consists of a B-tree storing M records with a branching factor $B=(logN)^{1/5}$ whose internal nodes are q-heaps.

Proof sketch: The previous literature has illustrated many examples of amortised optimisation algorithms which can also guarantee worst-case time , if their procedures are made somewhat more elaborate. We will use a similar approach here. Our algorithm will be in many respects analogous to [7]. The discussion will therefore be brief, and it may be helpful if the reader was previously acquainted with [7].

All the data will be stored at the leaf level of our tree and all leaves will have the same depth, as is conventional for B-trees. Say an internal node v is «safe» if its parity lies between B/4 and 3B/4, and it is «legal» if its parity lies between B/8 and B (in the special case of the root's parity, the previous lower bounds will equal 2). Our insertion/deletion algorithms will quarantee that each node has legal parity at all times , and it will attempt to make the parity safe as often as possible. In particular for fixed constant K>0, define ALG(K) to be a procedure that executes the following three steps after the insertion or deletion of a new leaf node.

1) If a deletion causes the root to have only one son then the procedure ALG(K) will simply eliminate the root and make its son the new root

2) If the parity of a node v increases and thereby becomes unsafe , then the algorithm will correct this problem by transferring one of the children of v to one of v's two adjacent siblings, provided this sibling is not made unsafe by this movement. If such a movement is impossible then the algorithm will start an 'evolutionary split process' that breaks v into two equal - sized nodes during the next K update commands which involve leaves descending from v. (This

evolutionary split operation will be designed to consume O(B/K) units of CPU time for 'fixing up v' during each of these K commands). The parent of v will view v as one child (rather than as 2) until the split evolution is completed.
(3) If the parity of a node v decreases and thereby becomes unsafe , then the algorithm solves this problem by transferring a child to v from one of its adjacent siblings provided that this movement does not cause the sibling to become unsafe. Otherwise , this operation will merge v with one of its two adjacent siblings by using a K-step evolutionary process, using O(B/K) CPU time per operation, analogous to the evolutionary split. Although v and its sibling will not be technically fully merged until the end of the K-evolutionary process, the parent of v will view them as one merged node at the onset of this evolutionary process, i.e. as one child rather than two. (The presence of two nodes temporarily representing an entity that should ideally form one object does not seriously degrade the performance because it will increase the time to probe a node by no more than a constant factor of 2).

The preceding algorithm is very similar in spirit to [7], and we will therefore give only a brief proof of its correctness. For all values of K>0 , it is easy to verify that the procedure ALG(K) will execute insertions and deletions in time O(h+hB/k) over a tree of height h. Moreover, if we set say K=B/128 , the algorithm will assure that the parity of all B-tree nodes will always be legal (causing the tree to have height h < O(logM/loglogN), and it will assure that update operations run in worst-case time O(h). Hence ALG(B/128) is a worst-case procedure that meets the claims of Lemma 2.

2.2 The Exponential Search tree

The Exponential Search tree answers queries in one-dimensional space. It is a multi-way tree where the degrees of the nodes decrease exponentially down the tree. Auxiliary information is stored in each node in order to support efficient searching queries. The Exponential Search tree has the following properties:

- Its root has degree $\Theta(N^{1/5})$.
- The keys of the root are stored in a local data structure. During a search, the local data structure is used to determine in which subtree the search is to be continued.
- The subtrees are exponential search trees of size $\Theta(N^{4/5})$.

The local data structure at each node of the tree is a combination of van Emde Boas trees and perfect hashing thus achieving O(logwloglogN) worst case cost for a search.
Anderson, by using an exponential search tree in the place of B-trees in the Fusion tree structure, avoids the need for weight-balanced trees at the bottom while at the same time improves the complexity for large word sizes. This structure is a significant improvement on linear space deterministic sorting and searching. On a unit-cost RAM with word size w, an ordered set of n w-bit keys (viewed as binary strings or integers) can be maintained in O(min ($\sqrt{}$logn, logn/logw + loglogn, logwloglogn)) time per operation, including insert, delete, member search, and neighbour search. The cost for searching is worst – case while the cost of updates is amortised. For range queries, there is an additional cost of reporting the found keys. As an application, n keys can be sorted in linear space at a worst-case cost O(n$\sqrt{}$logn). The best previous method for deterministic sorting and searching in linear space has been the fusion trees which supports queries in O(logn/loglogn) amortised time and sorting in O(nlogn/loglogn) worst – case time.

3 Algorithm Analysis

Theorem 1: Let $L_1,...,L_n$ be a set of n line segments in the plane. Then the set of all s pairwise intersections can be computed in time O((n+s)$\sqrt{}$logn) and space O(n).

Proof: We use plane sweep, i.e. we sweep a vertical line from left to right across the plane. At any point of the sweep we divide the set of line segments into three pairwise disjoint groups: dead, active, and dormant. A line segment is dead (active, dormant) if exactly two (one, zero) of its endpoints are to the left of the sweep line. Thus the active line segments are those which currently intersect the sweep line and the dormant line segments have not been encountered yet by the sweep. For the description of the algorithm we assume that no line segment is vertical and that no two endpoints or intersection points have the same x-coordinate. Both assumptions are made to simplify the exposition. The reader should have no difficulty to modify the algorithm such that it works without these assumptions.

The y-structure stores the active line segments ordered according to the y-coordinate of their intersection with the sweep line. More precisely, the y-structure is a Exponential search tree for the set of active line segments. In our example, line segments $L_1,L_2,...,L_6$ are active.

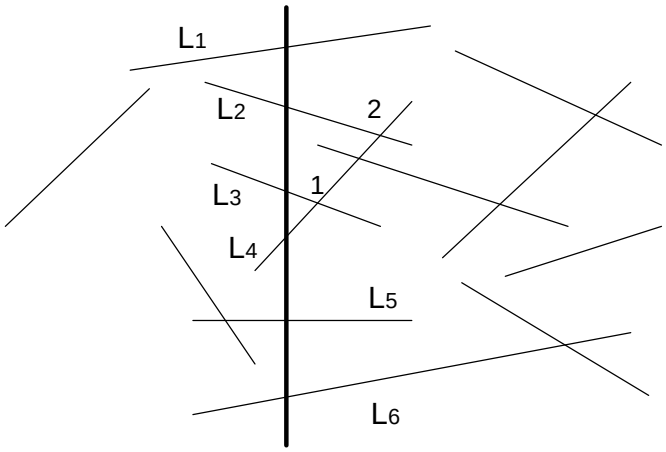

fig.2

They are stored in that order in the y-structure i.e. the y-structure is a dictionary for the set of active line segments. It is obvious that an Exponential search tree can be used as a dictionary. It supports (at least) the following operations in sub logarithmic time $O(\sqrt{\log n})$.

Find(p)	given point p on the sweep line, find the interval (on the sweep line) containing p
Insert(L)	insert line segment L into the y-structure
Delete(L)	delete line segment L from the y-structure
Pred(L), Succ(L)	find the immediate predecessor (successor) of line segment L in the y-structure
	Interchange(L,L') interchange adjacent line segments L and L' in the y-structure.

It is worthwhile to observe that the cost of operations Pred, Succ and Interchange can be reduced to O(1) under the following assumptions. First, the procedures are given a pointer to the leaves representing line segment L as an argument and second, additional pointers augment the tree structure. For the Pred and Succ operations we need pointers to the adjacent leaves and for the Interchange operation we need a pointer to the least common ancestor of leaves L and L'. Note that the least common ancestor of leaves L and L' contains the information which discriminates between L and L' in the tree search. Additional pointers do not increase the running time of Inserts or Deletes.

We describe the x-structure next. It contains all endpoints of line segments (dormant or active) which are to the right of the sweep line. Furthermore, it contains some of the intersections of line segments to the right of the sweep line. Note that it cannot contain all of them because the sweep has not even seen dormant line segments yet. The points in the x-structure are sorted according to their x-coordinate, so for the x-structure we use an other one Exponential search tree. For the correctness of the algorithm it is important that the x-structure always contains the point of intersections active line segments, which is closest to the sweep line. We achieve this goal by maintaining the following invariant.

If L_i and L_j are active line segments, are adjacent in the y-structure, and intersect to the right of the sweep line then their intersection is contained in the x-structure.

In our example, point 1 must be in the x-structure and point 2 may be in the x-structure. In the space-efficient version of the algorithm below point 2 is not in the x-structure. We have the following consequence of the invariant above.

Lemma 3: Let p be the intersection of active line segments L_i and L_j. If there is no endpoint of a line segment and no other point of intersection in the vertical strip defined by the sweep line and p then p is stored in the x-structure.

Proof: If p is not adjacent in the x-structure then L_i and L_j are not adjacent in the y-structure. Hence there must be active line segment L which is between L_i and L_j in the y-structure. Since L's right endpoint is not to the left of p either $\cap(L, L_i)$ or $\cap(L, L_j)$ is to the left of p, a contradiction.

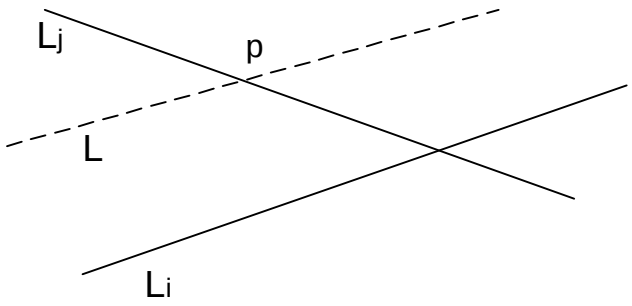

fig. 3

Finally, we maintain the following invariant about the output. All intersections of line segments, which are to the left of the sweep line, have been reported.

We are now in a position to give the details of the algorithm.

(1) y-structure $\leftarrow \varnothing$

(2) x-structure ← the 2n endpoints of the line segments sorted by x-coordinate
(3) while x-structure ≠ ∅
(4) do let p be a point with minimal x-coordinate in the x-structure
(5) delete p from the x-structure
(6) if p is a left endpoint of some segment L_j then
(7) search for p in the y-structure and insert L_j into the y-structure
(8) let L_i, L_k be the two neighbours of L_j in the y-structure;
insert $\cap(L_i, L_j)$ or $\cap(L_j, L_k)$ into the x-structure, if they exist;
(9) [delete $\cap(L_i, L_k)$ from the x-structure]
(10) fi;
(11) if p is a right endpoint of some segment L_j
(12) then let L_i, and L_k be the two neighbours of L_j in the y-structure;
(13) delete L_j from the y-structure;
(14) insert $\cap(L_i, L_k)$ into the x-structure if the intersection is to the right of the sweep line
(15) fi;
(16) if p is $\cap(L_i, L_j)$ then -- L_i, L_j are necessarily adjacent in the y-structure
(17) interchange L_i, L_j in the y-structure
(18) let L_h, L_k be the two neighbours of L_i, L_j in the y-structure
(19) insert $\cap(L_h L_j)$and $\cap(L_i, L_k)$ into the x-structure, if they are to the right of the sweep line;
(20) [delete $\cap(L_h, L_i)$ and $\cap(L_j, L_k)$ from the x-structure;]
(21) output p
(22) fi;
(23) od

In the algorithm above the statements in square brackets are not essential for correctness. Inclusion of these statements does not increase asymptotic running time, however it improves space complexity from O(n+s) to O(n).

It remains to prove correctness and to analyse the run time. For correctness it suffices to show that the invariants hold true. Call a point critical if it is either the endpoint of a line segment or an intersection of two line segments. Then the invariant about the x-structure and lemma 1 ensures that the point p selected in line (4) is the critical point, which is closest to and ahead of the sweep line. Thus every critical point is selected exactly once in line (4) and hence all intersections are output in line (21). Furthermore, line (7), (13), and (17) ensure that the y-structure always contains exactly the active line segments in sorted order and lines (8), (14), and (19) guarantee the invariant about the x-structure. In lines (9) and (20) we delete points from the x-structure whose presence is not required anymore by the invariant about the x-structure. This finishes the proof of correctness.

The analysis of the run time is quite simple. Note first that the, loop body is executed exactly 2n + s times, once for each endpoint and once for each intersection. Also a single execution deletes an element from a x-structure (that is an exponential search tree) in time $O(\sqrt{(\log(n+s))}) = O(\sqrt{\log n})$ since $s \leq n^2$ and performs some simple operations on a y-structure (that is an exponential search tree) of size n in $O(\sqrt{\log n})$ time Thus, the run time is $O((n+s)\sqrt{\log n})$.

The space requirement is clearly O(n) for the y-structure and O(n+s) for the x-structure. If we include lines (9) and (20) then the space requirement of the x-structure reduces to O(n) since only intersections of active line segments which are adjacent in the y-structure are stored in the x-structure with this modification. Thus space requirement is O(n).

4 Conclusions and Future Work

Lemma 4.
If the number of intersections S is $O(n\sqrt{\log n})$ our algorithm is optimal.

Proof: We must execute a comparison with algorithm [9] . So, we have to solve the following inequality:
$(n+s)\sqrt{\log n} < n\log n + s \Leftrightarrow n\sqrt{\log n} + s\sqrt{\log n} < n\log n + s \Leftrightarrow n(\log n - \sqrt{\log n}) > s(\sqrt{\log n} - 1) \Leftrightarrow n/s > (\sqrt{\log n} - 1) / (\log n - \sqrt{\log n}) \Leftrightarrow s/n < (\log n - \sqrt{\log n}) / (\sqrt{\log n} - 1)$
$\Leftrightarrow s < n (\log n - \sqrt{\log n}) / (\sqrt{\log n} - 1) = n(\sqrt{\log n}\sqrt{\log n} - \sqrt{\log n}) / (\sqrt{\log n} - 1) = n\sqrt{\log n}(\sqrt{\log n} - 1)/ (\sqrt{\log n} - 1) = n\sqrt{\log n} \Leftrightarrow s = O(n\sqrt{\log n})$.
Contrary to algorithm [9] that requires O(n+s) space, our algorithm requires O(n) space . Our future work includes the reduction of the search time of [9] algorithm using the RAM with W word size model.

References

[1] M. L. Fredman and D. E. Willard, "*Surpassing the information theoretic bound with fusion*

trees", J. Computer Systems Science 47, pp: 424-436, 1994.

[2] A. Anderson, "*Faster deterministic sorting and Searching in linear space*", TR- LU-Cs-TR:95-160, Department of Computer Science, Lund University, 1995.

[3] P. van Emde Boas, "*Preserving Order in a forest in less than logarithmic time and linear space*", IPL 6(3), 80-82, 1977.

[4] R. Raman, "*Improved data structures for predecessor queries in integer sets*", manuscript, 1995.

[5] Dan E. Willard, "*Applications of the Fusion Tree Method to Computational Geometry and Searching*" , ACM-SIAM symposium on discrete algorithms,pp:286-295, 1992

[6] D.E. Willard, "*Reduced Memory Space for Multi-Dimensional Search Trees*", 2-nd Symposium on Theoretical Aspects of Computer Science (published in Springer - Verlag LNCS182), 1985 , PP. 363-374. (The multibranching B-tree appears in the last section of this paper.)

[7] D.E. Willard and G.S.Lueker, "*Adding range restriction capability to dynamic data structures*", JACM 32 (1985) pp. 597-619.

[8] J.L. Bentley and T.A. Ottmann. "*Algorithms for reporting and counting geometric intersections*". IEEE Trans. Comput., C-28:643-647, 1979.

[9] B. Chazelle and H. Edelsbruner. "*An optimal algorithm for intersecting line segments in the plane*". J. Assoc. Comput. Mach., 39:1-54, 1992

[10] K.L. Clarkson and P.W. Shor. "*Applications of random sampling in computational geometry*", II. Discrete Comput. Geom., 4:387-421, 1989.