Scanline algorithms on a grid

Rolf G. Karlsson and Mark H. Overmars

RUU-CS-86-18
oktober 1986

Rijksuniversiteit Utrecht

Vakgroep informatica

Budapestlaan 6 3584 CD Utrecht
Corr. adres: Postbus 80.012 3508 TA Utrecht
Telefoon 030-53
The Netherlands

Scanline algorithms on a grid

Rolf G. Karlsson and Mark H. Overmars

RUU-CS-86-18
oktober 1986

Department of Computer Science
University of Utrecht
P.O. Box 80.012
3508 TA Utrecht
the Netherlands

SCANLINE ALGORITHMS ON A GRID

by

Rolf G. Karlsson[1] and Mark H. Overmars[2]

[1] Dept. of Computer Science, Linköping University, 581 83 Linköping, Sweden

[2] Dept. of Computer Science, University of Utrecht, P.O. Box 80.012, 3508 TA Utrecht, The Netherlands

Abstract: A number of important problems in computational geometry are solved efficiently on 2- or 3-dimensional grids by means of scanline techniques. In the solutions to the maximal elements and closure problems, a factor $\log n$ is substituted by $\operatorname{loglog} u$, where n is the set size and u the grid size. Next, by using a data structure introduced in the paper, the interval trie, previous solutions to the rectangle intersection and connected component problems are improved upon. Finally, a fast intersection finding algorithm for arbitrarily oriented line segments is presented.

Keywords and phrases: scanline techniques, maximal elements, rectangle problems, connected components, line segment intersection.

1. Introduction

Computational geometry studies the computational complexity of finite geometric problems. Recently, there has been a growing interest in solving problems in computational geometry on a grid, i.e., points (or line segments with endpoints) in $U^d = [0..u\text{-}1]^d$. See for example, Karlsson [7], Karlsson and Munro [8], Keil and Kirkpatrick [10], Müller [15], Overmars [16], and Willard [20,21]. In some sense, computational geometry on a grid shows the complexity of a problem after sorting. But apart from being of great theoretical interest, it is our belief that structures on grids are potentially practical. Indeed, analysis of algorithms with a restricted domain is of interest for real applications since bounds on the key space for a particular instance of a problem are often known. The methods we present should be useful within computer graphics, VLSI design, and robotics. In graphics, for instance, the key space is a moderate sized raster and, hence, all points are in U^d for some small U. In this paper we mainly deal with 2-dimensional ($d = 2$) orthogonal problems (line segments are axis-parallel). VLSI technology, for instance, often uses only orthogonal object boundaries and wires. Note that, with appropriate translation and scaling, U^2 can approximate any bounded region of the real plane.

On a grid many problems can be solved more efficiently by using perfect hashing techniques or table look-up, but due to their costly preprocessing such methods will not do for the problems we consider. Instead, we achieve our fast solutions by a more moderate preconditioning work, and by scanning the n points or objects along one axis while maintaining appropriate data structures.

We demonstrate the technique by applying it to the following problems:

(i) **Maximal elements:** Given a set of points on a grid, determine those points that are maximal, i.e. in 2 dimensions, the points $p = (x_p, y_p)$ for which there are no other points $q = (x_q, y_q)$ such that $x_p \leq x_q$ and $y_p \leq y_q$.

(ii) **Maximal layers:** Given a set of points on a grid, determine the successive layers of maximal elements.

(iii) **Orthogonal convex hull:** Given a set of points on a grid, compute the orthogonal convex hull, i.e., the smallest convex region that contains all the points, and where convexity is defined through axis-parallel point connections.

(iv) **Closure:** Given a set of orthogonal rectangles on a grid, find the closure of their union.

(v) **Rectangle intersection:** Given a set of orthogonal rectangles on a grid, determine all pairs of intersecting rectangles. Two types of intersection are considered: boundary intersection in which two rectangles intersect if their boundaries intersect and true intersection in which two rectangles intersect if they have some point in their interior in common.

(vi) Connected component: Given a set of orthogonal rectangles on a grid, determine their connected components, where two or more rectangles are said to be connected whenever each two points in their union are connected.

(vii) Line segment intersection: Given a set of arbitrarily oriented line segments with endpoints on a grid, report all intersections.

See the following table for the results obtained (logarithms to the base 2 when none is specified).

Problem	Known bound	New bound
3-dim maximal elements	$n\log n$ [12]	$n\log\log u$
maximal layers	$n\log n$	$n\log\log u$
orthogonal convex hull	$n\log n$ [e.g. 5]	$n\log\log_n u$
closure	$n\log n$ [18]	$n\log\log u$
boundary intersection	$k + n\log n$ [3,13]	$k + n\log\log u$
true intersection	$k + n\log n$ [3,13]	$k + n\log u/\log\log u$
connected components	$n\log n$ [4]	$n\log u/\log\log u$
line segment intersection	$k + n\log^2 n/\log\log n$ [2] or $(k+n)\log n$ [1]	$k + u + n\log n$ or $(k+n)\log\log u + n\log n$

To solve some of these problems a new dynamic 1-dimensional structure for the stabbing problem (for a set of intervals, determine those containing a query point) on a grid is designed that is more efficient than known dynamic solutions. We will call this structure an interval trie.

The structure of the paper is as follows. In Section 2 we describe the interval trie and give some preliminaries that will be used in the rest of the paper. In Section 3 we demonstrate the scanline technique by applying it to the maximal elements and closure problems. In Section 4 we also invoke the interval trie to solve rectangle intersection problems. In Section 5 we solve the connected components problem. In Section 6 we allow line segments to be arbitrarily oriented when solving the intersection problem. Two new solutions will be given. Finally, Section 7 contains conclusions and some open problems.

2. Preliminaries

Let us first recall some known results that will be used throughout this paper. Van Emde Boas [19] presented a structure which can store n keys from $U = [0..u-1]$ using $O(u)$ space so that in time $O(\log\log u)$ keys can be inserted, deleted and the key nearest to a specified key in U can be found. A drawback of this structure is its

$\Theta(u)$ preprocessing. Instead, we will use a more flexible tree structure introduced by Johnson.

Theorem 2.1: (Johnson [6]) Given a set of n keys from $U = [0..u-1]$, there exists a structure using $O(u)$ storage such that in time $O(\text{loglog } u)$ it can be initialized and keys can be inserted, deleted and the key nearest to a given value in U can be determined.

We will call this structure a Johnson tree. It can also be used to solve 1-dimensional range queries in $O(k+\text{loglog } u)$ time, where k is the number of reported answers.

In solving problems we will presort the given points. It is well known, that given a set of n keys in U, they can be sorted in $O(n+u)$ time using $O(n+u)$ storage. However, it is possible to do it more efficiently.

Theorem 2.2: (Kirkpatrick and Reisch [11]) Given a set of n keys in U, they can be sorted in $O(n(1+\text{loglog}_n u))$ time using $O(n+u)$ storage.

We will now describe a new dynamic data structure, called the interval trie, used to solve the 1-dimensional stabbing problem on a grid, that is, given a set of intervals $[a_i..b_i]$ with a_i and b_i in U, store them so that for any query point p in U we can efficiently determine the intervals that contain p. Let $F(u) = \log^\epsilon u$ for some $\epsilon > 0$. We split the set of intervals V into sets $V_1, V_2, \ldots$ where V_i contains all segments of length between $F(u)^{i-1}$ and $F(u)^i$. Clearly, there are at most $O(\log u/\log F(u)) = O(\log u/\text{loglog } u)$ sets V_i. We treat the V_i's separately. For each V_i we divide the universe in equal parts of size $F(u)^{i-1}$. Hence, each interval has its beginpoint in one part, overlaps at most $F(u)$ parts, and has its endpoint in another part. No interval can have begin and endpoint in the same part. For each part we have three lists: B of beginpoints, E of endpoints, and O of overlapping intervals. Both B and E we store as Johnson trees making it possible to insert and delete begin and endpoints in $O(\text{loglog } u)$ time. We connect the begin and endpoints in a double linked list. The set O we store as a list. Moreover, we build a global Johnson tree S that stores for each interval where it is stored in lists O.

To perform a query we query each V_i separately. In V_i we find the part containing the query point p in $O(1)$ time. We report all intervals in the corresponding list O and report the correct intervals in B and E by walking along the lists in the correct order. In list B we start at the begin point that lies leftmost. If it lies left of p its interval must contain p and we report it. After this we look at the next beginpoint in the same way. We continue until we find a beginpoint that lies to the right of p. In E we walk along the points from right to left in a similar way. Hence, per V_i we spend time proportional to the number of answers. It is easy to see that in this way all answers are reported exactly once.

To insert and delete an interval we determine the V_i the interval belongs to.

We insert or delete it in the right B and E structures, insert or delete it in S and add or remove it from at most $O(F(u))$ O lists (updating and using S).

Theorem 2.3: Given a set of n intervals on a grid U, for each $\epsilon > 0$ there exists a dynamic structure for solving the stabbing problem with a query time of $O(\log u/\log\log u)$, an insertion and deletion time of $O(\log^{\epsilon} u)$, and using $O(u\log^{\epsilon} u)$ storage.

Proof: Follows from the above discussion. □

It should be noticed that constructing the structure for an empty set takes time $O(\log\log u \log^{\epsilon} u)$ time.

3. Maximal Elements and Closure

We will now demonstrate the scanline technique by applying it to the 2-dimensional maximal elements problem, where a point $p = (x_p, y_p)$ is said to be maximal if there are no points $q = (x_q, y_q)$ such that $x_p \leq x_q$ and $y_p \leq y_q$. After an initial lexicographical sorting of the points, using the algorithm in Theorem 2.2, we scan the x-axis from right (value u-1) to left (value 0), stopping at the x-coordinate of each point in the set. With the scanline we keep track of the point with highest y-coordinate passed so far. (In the beginning this y-coordinate is initialized to 0.) If the y-value at the current point is larger than the value stored with the scanline, the current point must be maximal. Hence, we can report it and store its y-value at the scanline. In this way we find the set of maximal elements in $O(n(1 + \log\log_n u))$ time.

By maintaining a Johnson tree to store the current y-values of successive layers of maximal elements during the scan, we can similarly solve the maximal layers problem. When stopping at a point p we can in $O(\log\log u)$ time determine the y-value y_L of the nearest layer L below p. We update the tree by deleting y_L and inserting y_p to form the continuation of L. If the point p is smaller than any y-value in the tree it starts a new layer and is inserted in the tree.

Theorem 3.1: Given n points from U^2 we can compute the maximal layers in time $O(n\log\log u)$ using $O(u + n)$ storage.

Proof: Follows from the presentation above. □

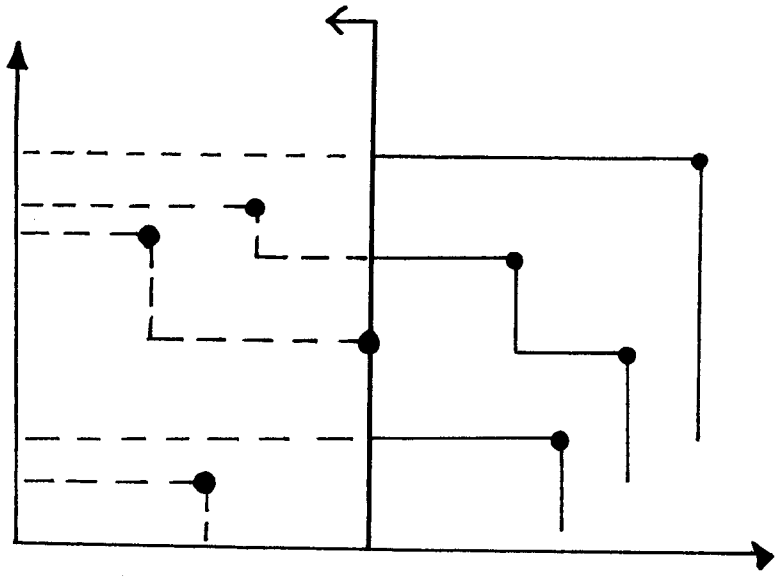

Figure 1. Forming maximal layers by scanning points from right to left

Similarly, a plane sweep technique gives an $O(n\text{loglog}u)$ time solution to the 3-dimensional maximal elements problem, as follows.

Theorem 3.2: Given n points from U^3 we can compute the maximal elements in time $O(n\text{loglog } u)$ using $O(u+n)$ storage.

Proof: We first sort all the points with respect to their z-coordinate. This takes $O(n(1+\text{loglog}_n u))$ time and $O(n+u)$ storage, by Theorem 2.2. Next we initialize an empty Johnson tree in $O(\text{loglog } u)$ time. We then sweep a plane that is parallel to the xy-plane in the z-direction from $u-1$ to 0, stopping at each point we pass. With the plane we keep the contour of maximal elements of the projections of the points we have passed. We store in the Johnson tree the maximal points on the boundary of the projected xy-contour area, ordered according to x-coordinate. When the scan plane passes a point, p, we update the Johnson tree, by checking whether p lies above or below the current contour. After finding the closest point to p in the tree, we can in $O(1)$ time decide if p lies outside or inside the projected contour area. If it lies outside it is a new maximal point and we report it, and augment the contour. Otherwise p is not maximal and we can proceed to the next point. It is easy to see that in this way all points that are maximal will be correctly reported. An update of the Johnson tree may involve deleting many points, but counted over all updates each point may only be inserted and deleted once. Thus, moving the scanplane and performing the queries and updates takes $O(n\text{loglog } u)$ time in total. □

The above algorithms we adapt to solve two other problems, finding the orthog-

onal convex hull and computing the closure. The convex hull of a set is the smallest convex region that contains the set. The prefix orthogonal means that convexity is defined by axis-parallel point connections, i.e., for each two points in the polygon on a horizontal or vertical line, the line segment in between them lies completely inside the polygon. Keil and Kirkpatrick [10] solve the similar problem of computing the non-orthogonal convex hull in the same time bound.

Theorem 3.3: Given n points from U^2 we can compute their orthogonal convex hull in time $O(n(1 + \log\log_n u))$ using $O(u + n)$ storage.

Proof: We observe the following. By determining the four extreme x- and y-values of the point set, the orthogonal convex hull is given by four monotonic staircase curves between the extreme points. Thus, we solve our problem by computing the maximal elements, changing the definition of maximality appropriately between each of the four extrema. □

Next we consider the following problem. Given a set of n axis-parallel rectangles, compute their closure, defined as follows. Two points $p_1 = (x_1, y_1)$ and $p_2 = (x_2, y_2)$ are *incomparable* if neither one dominates the other (they are mutually maximal). Assume WLOG $x_2 > x_1$ and $y_2 < y_1$; then the *SW-conjugate* and *NE-conjugate* of p_1 and p_2 are $q_1 = (x_1, y_2)$ and $q_2 = (x_2, y_1)$ respectively. A region R is *NE-closed* if for any two incomparable points p_1 and p_2 *connected* in R, the NE-conjugate of p_1 and p_2 is also in R. Analogously, we define a SW-closed region. Based on this, the *NE-closure* of a region S of the plane, denoted $NE(S)$, is the smallest NE-closed region containing S. Analogously, we define the SW-closure. The *closure* is the smallest region containing S that is both SW-closed and NE-closed.

A reason for considering the closure problem, is that it has an important relation to the problem of safeness and deadlock-freedom when several users concurrently access a database [22]. There, it is common for a user to lock a variable, for exclusive access, at the beginning of an update and unlock it afterwards. If the two axes in our problem formulation reflect the history of actions taken by two users, then a rectangle depicts a time slot when both users would access a variable. Given this, one can show that any sequence of lock, update and unlock steps which avoids the SW-closure is deadlock-free. Using our technique we speed up the $O(n\log n)$ time closure algorithm, given by Soisalon-Soininen and Wood [18].

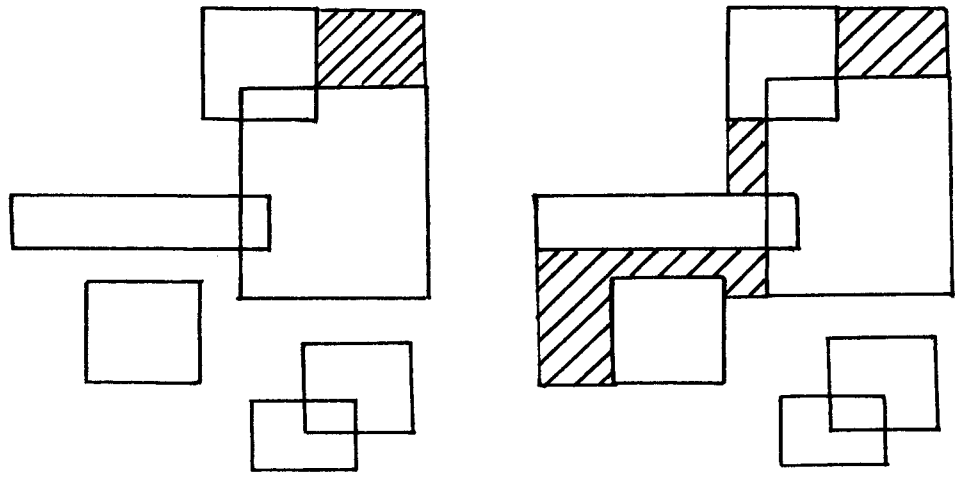

Figure 2.(a) The NE-closure, and (b) the closure of a set of rectangles

Theorem 3.4: Given n rectangles with corners from U^2, we can compute their closure in time $O(n\log\log u)$ using $O(n+u)$ storage.

Proof: Soisalon-Soininen and Wood [18] showed that the closure can be determined by first computing the NE-closure, through a left-to-right scan of the rectangles, followed by a right-to-left scan to get the SW-closure of the result. We outline their algorithm as described in [17]. For each connected component, the boundary of its closure consists of two x-monotonic staircase curves. A component is called *active* if it is intersected by the scanline. Thus, a component is active if and only if it contains at least one (active) rectangle intersecting the scanline. An active component is characterized by an active interval, which is the segment between the maximum y-value of all rectangles found so far in the component and the minimum y-value of the currently active rectangles. The scanline will stop at either a left side or a right side of a rectangle. Consider the left-to-right scan that constructs the NE-closure (the SW-closure of the result is constructed analogously).

At the left side of a rectangle, R, we either (i) initialize an active interval (when R does not overlap any active component), or (ii) extend an active interval (when R overlaps only one active component), or (iii) merge two or more active intervals with R.

When the scanline stops at the right side of a rectangle, R, either (i) R is deactivated shrinking its active interval upward (when R is the bottommost active rectangle of the component), or (ii) the component is terminated (when R is the only active rectangle of the component), or (iii) R is deactivated without affecting the active interval.

To handle the above actions, we store two types of elements in a Johnson tree, T, the lower endpoints of the active intervals and the lower y-values of active rectangles. These elements are linked in two leaf lists. We note that the component set of active rectangles are disjoint. Thus, by linking the leaves which store lower y-values we know the rectangle order and cardinality within each component. This makes merging of active intervals easy.

When encountering a rectangle left side $[y_1..y_2]$, the greatest interval endpoint smaller than y_1 is found in T and its leaf list is traversed until also y_2 is located. This takes time $O(m+ \text{ loglog } u)$, where m is the number of active intervals that will be merged by the new rectangle. Clearly, two active intervals can be merged and their lists of active rectangles concatenated in $O(\text{loglog } u)$ time. By consistently charging the cost of merging two active intervals to the rightmost rectangle involved, the total time is $O(n\text{loglog } u)$ since each rectangle can be charged at most once. The updating of T when encountering a right side follows similarly from cases i) to iii) above. We can in $O(\text{loglog } u)$ time decide if an active rectangle is the bottommost, or the last active in a component. □

4. Rectangle Intersection

In this section, we present solutions to two variants of the rectangle intersection problem. First, we only look for rectangles which intersect each other's edges. We call this boundary intersection.

Theorem 4.1: Given n orthogonal rectangles on U^2, the pairs of rectangles that boundary intersect can be located in time $O(k + n\text{loglog } u)$ using $O(n + u)$ space, where k is the number of reported answers.

Proof: We can also state the problem as follows: Given a set of horizontal and vertical line segments, report all intersections. To solve this problem we use a plane sweep approach. We sweep a scanline from left to right. With the scanline we keep a Johnson tree, T, storing the y-coordinates of all horizontal line segments intersecting the scanline. When the scanline encounters the left endpoint of a horizontal line segment, its y-coordinate is inserted into T; when it is a right endpoint its y-coordinate is deleted from T. These updates are done in $O(n\text{loglog } u)$ total time. When the scanline encounters a vertical line segment, $I = [y_1..y_2]$, a range query is performed. We locate the greatest y-coordinate in T smaller than or equal to y_2, and the smallest y-coordinate greater than or equal to y_1. By traversing the linked list between these two leaves, we find the horizontal segments intersecting I. It should be clear that in this way all intersections are reported in the stated time bound. □

Next, we extend the algorithm to also handle proper rectangle containment, in which two rectangles intersect if they have some point in their interior in common, e.g., one rectangle can lie completely inside the other.

Theorem 4.2: Given n orthogonal rectangles on U^2, the pairs of rectangles that true intersect can be located in time $O(k + n\log u/\log\log u)$ using $O(n + u\log^{\epsilon} u)$ space, where k is the number of reported answers.

Proof: Rectangles R_1 and R_2 intersect if their boundaries intersect or the left bottom corner of R_1 lies in R_2 or the left bottom corner of R_2 lies in R_1. More than one can happen, but with some care it is possible to avoid that intersections are reported more than once. Hence, we can solve the problem by solving two problems: the boundary intersection problem and the following problem: Given a set of points and a set of rectangles, determine for each rectangle what points it contains. The boundary intersection can be solved as described in Theorem 4.1. To solve the second problem we again sweep a scanline from left to right. With the scanline we keep an interval trie containing the y-extensions of all rectangles that intersect the scanline. When the scanline encounters a left edge of a rectangle it is interserted in the interval trie. When we encounter a right edge it is deleted. When the scanline encounters a point we perform a stabbing query with this point on the interval trie. By Theorem 2.3, this will report rectangle containments in the stated time bound. □

5. Connected Components

We consider the following problem. Given n axis-parallel rectangles, compute the set of connected components. Note this problem is more complicated than finding the closure, where rectangles are merged to form monotonic components. To solve the connected component problem we again sweep a line from left to right over the plane. With the scanline, we maintain three data structures. First, a cluster trie, storing the connected components; secondly, an endpoint trie, storing the endpoints of the active rectangles (rectangles intersected by the scanline); and finally, an interval trie, storing the y-intervals of active rectangles. This set-up is somewhat similar to the one in [4], but the new data structures provide a more efficient solution.

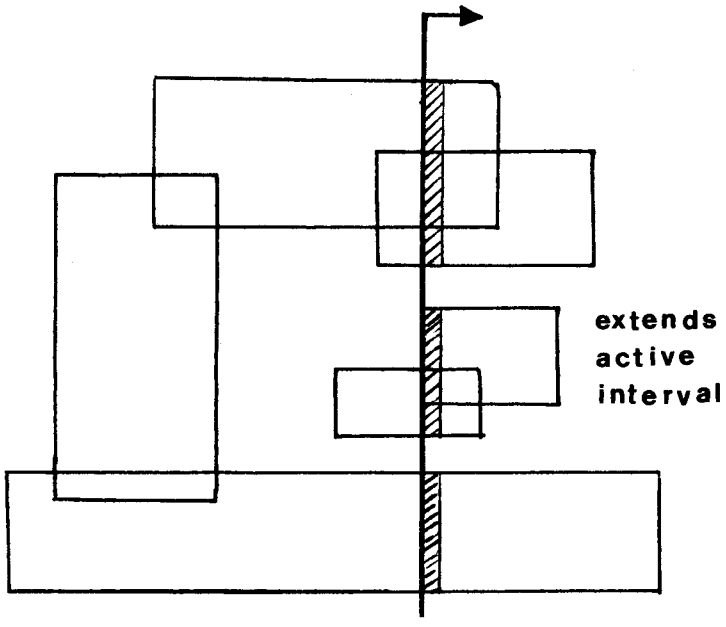

Figure 3. Scanning for connected components (active intervals are shaded)

Theorem 5.1: Given n orthogonal rectangles on U^2 we can compute the connected components in time $O(n\log u/\log\log u)$ using $O(n + u\log^{\epsilon} u)$ storage.

Proof: Each connected component has a cluster identifier, and is represented by one or more active intervals, defined as follows. An active interval is a segment between the minimum and maximum y-value of a non-interrupted subset of active rectangles in the component (i.e., the segment should not overlap rectangles from other clusters). We implement the cluster trie and the endpoint trie as Johnson trees. When the scanline encounters the left side of a rectangle, the y-interval of the rectangle is inserted in the interval trie and the corresponding endpoints are inserted in the endpoint trie; when a right side is encountered the y-interval and the endpoints of the rectangle are deleted. The cluster trie is updated as follows.

When inserting an interval $I = [y_1..y_2]$ in the interval trie, we can in the cluster trie, in $O(\log\log u + m)$ time, find the m active intervals overlapping I. Next, we determine if I true intersects at least one other rectangle, R. If $m > 1$, this is immediate; otherwise we search the endpoint trie to see whether I contains an endpoint of another y-interval and if not, search the interval trie top-down for an interval containing I. This takes $O(\log u/\log\log u)$ time. Having found that I true intersects a rectangle R, the active interval, A, including R is extended in the cluster trie if y_1 is smaller than the minimum value of A or y_2 is greater than the maximum value of A. In general, when I intersects more than one active interval, we amortize the time to merge them (like in the closure solution, Theorem 3.4) by charging the rightmost rectangle involved in a two-interval merge with the

$[x..x+1)$. We will take care that the INTER bags only contain intersections between neighbors in the tree T (see below). To initialize the structures, each line segment is stored in the proper BEGIN and END bag, and all INTER bags are made empty. Next, we move a scanline from left to right over the plane. With the scanline we keep a balanced binary tree T (e.g. an AVL tree) storing the line segments that are intersected by the scanline. The scanline stops at each integer x-coordinate x. At a stop position x we do the following three things:

(1) Insert all line segments in BEGIN(x) in T. Compute the intersections at the inserted segments with their neighbors and insert them in the proper INTER bags. For each pair of line segments that stop being neighbor (because a new segment was inserted in between them), remove them from the right INTER bag. (This can be done in O(1) time per line segment inserted when we keep with each line segment in T pointers to the position in INTER bags of its intersections with neighbors.)

(2) Delete all line segments in END(x) from T. This means that new pairs of line segments in T become neighbors. Compute the intersections between these line segments and insert them in the right INTER bags.

(3) For each intersection in INTER(x), report it and change the position of the two intersecting segments in T. This means that some new neighbors appear in T and some old ones disappear. Insert and remove the appropriate intersections from the INTER bags. Since this might add new intersections to INTER(x), we have to check that they have not already been reported. As we do not treat intersections in left to right order (because INTER is an unordered bag) this is no trivial operation. To check whether the intersection has already been reported we adapt a technique from Myers [15]. Assume we want to add the intersection between s and s' to INTER(x). We now check in which order s and s' occur at position $x+1$. If they are already in this order their intersection must already have been treated and, hence, nothing needs to be done. Otherwise we insert the intersection in INTER(x). This clearly takes only O(1) time.

Some intersections removed from INTER(x) may not yet have been treated, but are anyway guaranteed to reappear. It is clear that in this way the intersections in INTER(x) are *not* performed in a left to right order! It just depends on the order in which they are in the bag. We claim this does not matter and that all intersections between x and $x+1$ will be properly reported. This follows from two observations. (i) When an intersection is reported, it does exist and has not been reported before. (ii) When INTER(x) is empty, this means that each two neighbors in T either do not intersect between x and $x+1$ or that their intersection has been reported. We will show that this can only be the case when the line segment in T are in the right order, i.e., in the order they should be in immediately before $x+1$. When they are in this order, all intersections must have been reported

because each segment must have changed place with all the segments with which it intersect between x and $x+1$. To prove that the line segments are in the right order, assume they are not. Then there must be two neighbors l_1 and l_2 that should be in the reverse order. Assume l_1 and l_2 do not intersect between x and $x+1$. In this case they cannot have changed place between x and $x+1$ and thus are in the right order. Assume l_1 and l_2 do intersect between x and $x+1$. As they are neighbors and INTER(x) is empty, their intersections must have been reported and, hence, they must have changed place. As it is impossible that they again changed place, their order is correctly the reverse of the order at position x. This proves that l_1 and l_2 are in the right order. Contradiction. Hence, all line segments in T are indeed in the right order. This shows that the algorithm correctly reports all intersections.

What remains to be shown is the time bound. Initializing the tree T and the bags is O(1) work, while sorting the begin and endpoints along x-coordinate and putting them in the right bags takes $O(n(1 + \text{loglog}_n u))$ time. Inserting and deleting a segment in T requires O(log n) work, or O($n$log n) counted over all insertions and deletions. Performing an intersection takes O(1) time, assuming that with each intersection we store a pointer to the right position in the tree T, and also the check whether an intersection has been reported before takes O(1) time. Unfortunately, we have to stop at each x-coordinate to see whether one of the three bags contains points. This adds a term O(u) to the time bound. Given k reported intersections, the time bound follows.

The storage required for the bags and the tree T is O($n+u$) because only intersections between neighbors are stored. □

When u is small this method is an improvement over known techniques, but when u grows the method soon becomes worse. The O(u) time is only spent because for each x-position we have to check whether one of the bags is not empty. To avoid this, we store the x-positions with non-empty bags in a Johnson tree. In this way we can find the next non-empty bag in time O(loglog u). Initially, all positions at which a line segment starts or ends are stored in this tree. When an intersection is found that has to be put in bag INTER(x), we search whether x is in the tree. If it is, we just put the intersection in INTER(x). Otherwise, we first insert x in the tree. In this way the tree is maintained correctly.

Theorem 6.2: Given n line segments on U^2, we can compute all k intersections in time O($(k+n)$loglog u + $n$log n) using O($n+u$) storage.

Proof: Follows from the above discussion, since a Johnson tree is initialized in O(loglog u) time. □

7. Conclusions and Open Problems

We have applied scanline techniques to solve a number of important problems in computational geometry on a grid. As an algorithmic tool in the solutions, we have introduced the interval trie, a grid correspondent to the segment tree. For typical values on grid size and set size, say in computer graphics, our bounds are better than the known bounds.

We believe that many other geometry problems can be solved efficiently by using the methods in this paper. Problems like the measure problem and other contour problems are presently being looked at. Furthermore, since there exist scanline algorithms for triangulation and construction of Voronoi diagram it would be interesting to see if the methods of this paper can be employed to make such solutions faster.

In another paper, we have extended some of the results in this paper to higher dimensions [9].

8. References

[1] J.L. Bentley and T. Ottman, Algorithms for Reporting and Counting Geometric Intersections, *IEEE Trans. Comp. C-28*, 9 (1979), 643-647

[2] B. Chazelle, Intersecting is Easier than Sorting, *Proc. 16th Annual ACM Symposium on Theory of Computing* (1984), 125-134

[3] H. Edelsbrunner, A New Approach to Rectangle Intersections, Part II, *Int. J. Comput. Math. 13* (1983), 221-229

[4] H. Edelsbrunner, J. van Leeuwen, T. Ottmann and D. Wood, Computing the Connected Components of Simple Rectilinear Geometrical Objects in d-space, *R.A.I.R.O. Theoretical Informatics 18*, 2 (1984), 171-183

[5] R.A. Jarvis, On the Identification of the Convex Hull of a Finite Set of Points in the Plane, *Information Processing Lett. 2* (1973), 18-21

[6] D.B. Johnson, A Priority Queue in which Initialization and Queue Operations Take O(loglog D) Time, *Math. Systems Theory 15*, 4 (1982), 295-310

[7] R.G. Karlsson, Algorithms in a Restricted Universe, Ph.D. thesis, University of Waterloo, 1984, Dept of Computer Science Tech. Report CS-84-50

[8] R.G. Karlsson and J.I. Munro, Proximity on a Grid, *Proc. 2nd Symposium on Theoretical Aspects of Computer Science*, Springer-Verlag Lecture Notes in Computer Science 182 (1985), 187-196

[9] R.G. Karlsson and M.H. Overmars, Normalized Divide and Conquer, to appear

[10] J.M. Keil and D.G. Kirkpatrick, Computational Geometry on Integer Grids, *Proc. 19th Annual Allerton Conference* (1981), 41-50

[11] D. Kirkpatrick and S. Reisch, Upper Bounds for Sorting Integers on Random Access Machines, *Theoretical Computer Science 28* (1984), 263-276

[12] H.T. Kung, F. Luccio and F.P. Preparata, On Finding the Maxima of a Set of Vectors, *J. ACM 22*, 4 (1975), 469-476

[13] E.M. McCreight, Efficient Algorithms for Enumerating Intersecting Intervals and Rectangles, Xerox Palo Alto Res. Center, Tech. Report PARC CSL-80-9, 1980

[14] F.W. Myers, An $O(E \log E + I)$ Expected Time Algorithm for the Planar Segment Intersection Problem, *SIAM J. Computing 14* (1985), 625-637

[15] H. Müller, Rastered Point Location, *Proc. Workshop on Graphtheoretic Concepts in Computer Science* (WG85), Trauner Verlag, 1985, 281-293

[16] M.H. Overmars, Range Searching on a Grid (ext. abst.), *Proc. Workshop on Graphtheoretic Concepts in Computer Science*(WG85), Trauner Verlag, 1985, 295-305

[17] F.P. Preparata and M.I. Shamos, *Computational Geometry, An Introduction*, Springer-Verlag, 1985

[18] E. Soisalon-Soininen and D. Wood, An Optimal Algorithm for Testing for Safety and Detecting Deadlock in Locked Transaction Systems, *Proc. ACM Symposium on Principles of Data Bases* (1982), 108-116

[19] P. van Emde Boas, Preserving Order in a Forest in Less Than Logarithmic Time and Linear Space, *Information Processing Lett. 6*, 3 (1977), 80-82

[20] D.E. Willard, Log-logarithmic Worst-case Range Queries Are Possible in Space $\Theta(n)$, *Information Processing Lett. 17*, 2 (1983), 81-84

[21] D.E. Willard, New Trie Data Structures Which Support Very Fast Search Operations, *J. Comput. Syst. Sci. 28* (1984), 379-394

[22] M.Z. Yannakakis, C.H. Papadimitriou and H.T. Kung, Locking Policies: Safety and Freedom for Deadlock, *Proc. 20th Annual IEEE Symposium on Foundations of Computer Science* (1979), 286-297